Nominal Unification

Christian Urban [a] Andrew Pitts [a] Murdoch Gabbay [b]

[a] *University of Cambridge, Cambridge, UK*

[b] *INRIA, Paris, France*

Abstract

We present a generalisation of first-order unification to the practically important case of equations between terms involving *binding operations*. A substitution of terms for variables solves such an equation if it makes the equated terms α*-equivalent*, i.e. equal up to renaming bound names. For the applications we have in mind, we must consider the simple, textual form of substitution in which names occurring in terms may be captured within the scope of binders upon substitution. We are able to take a "nominal" approach to binding in which bound entities are explicitly named (rather than using nameless, de Bruijn-style representations) and yet get a version of this form of substitution that respects α-equivalence and possesses good algorithmic properties. We achieve this by adapting two existing ideas. The first one is terms involving explicit substitutions of names for names, except that here we only use *explicit permutations* (bijective substitutions). The second one is that the unification algorithm should solve not only equational problems, but also problems about the *freshness* of names for terms. There is a simple generalisation of classical first-order unification problems to this setting which retains the latter's pleasant properties: unification problems involving α-equivalence and freshness are decidable; and solvable problems possess most general solutions.

Key words: Abstract Syntax, Alpha-Conversion, Binding Operations, Unification

1 Introduction

Decidability of unification for equations between first-order terms and algorithms for computing most general unifiers form a fundamental tool of computational logic with many applications to programming languages and computer-aided reasoning. However, very many potential applications fall outside the scope of first-order unification, because they involve term languages with binding operations where at the very least we do not wish to distinguish terms differing up to the renaming of bound names. There is a large body of work studying languages with binders through the use of various λ-calculi as term representation languages, leading to

Accepted for publication in Theoretical Computer Science.

higher-order unification algorithms for solving equations between λ-terms modulo $\alpha\beta\eta$-equivalence. However, higher-order unification is technically complicated without being completely satisfactory from a pragmatic point of view. The reason lies in the difference between substitution for first-order terms and for λ-terms. The former is a simple operation of textual replacement (sometimes called *grafting* [1], or *context substitution* [2, Sect. 2.1]), whereas the latter also involves renamings to avoid capture. Capture-avoidance ensures that substitution respects α-equivalence, but it complicates higher-order unification algorithms. Furthermore it is the simple textual form of substitution rather than the more complicated capture-avoiding form which occurs in many informal applications of "unification modulo α-equivalence". For example, consider the following schematic rule which might form part of the inductive definition of a binary evaluation relation $\Downarrow$ for the expressions of an imaginary functional programming language:

$$\frac{\mathtt{app}(\mathtt{fn}\, a.Y, X) \ \Downarrow \ V}{\mathtt{let}\, a = X \,\mathtt{in}\, Y \ \Downarrow \ V} \ . \tag{1}$$

Here X, Y and V are metavariables standing for unknown programming language expressions. The binders $\mathtt{fn}\, a.(-)$ and $\mathtt{let}\, a = X \ \mathtt{in}\ (-)$ may very well capture free occurrences of the variable named a when we instantiate the schematic rule by replacing the metavariable Y with an expression. For instance, using the rule scheme in a bottom-up search for a proof of

$$\mathtt{let}\, a = 1 \,\mathtt{in}\, a \ \Downarrow \ 1 \tag{2}$$

we would use a substitution that does involve capture, namely

$$[X := 1, Y := a, V := 1]$$

in order to unify the goal with the conclusion of the rule (1)—generating the new goal $\mathtt{app}(\mathtt{fn}\, a.a, 1) \Downarrow 1$ from the hypothesis of (1). The problem with this is that in informal practice we usually identify terms up to α-equivalence, whereas textual substitution does not respect α-equivalence. For example, up to α-equivalence, the goal

$$\mathtt{let}\, b = 1 \,\mathtt{in}\, b \ \Downarrow \ 1 \tag{3}$$

is the same as (2). We might think (erroneously!) that the conclusion of rule (1) is the same as $\mathtt{let}\, b = X \,\mathtt{in}\, Y \Downarrow V$ without changing the rule's hypothesis—after all, if we are trying to make α-equivalence disappear into the infrastructure, then we must be able to replace any *part* of what we have with an equivalent part. So we might be tempted to unify the conclusion with (3) via the textual substitution $[X := 1, Y := b, V := 1]$, and then apply this substitution to the hypothesis to obtain a wrong goal, $\mathtt{app}(\mathtt{fn}\, a.b, 1) \Downarrow 1$. Using λ-calculus and higher-order unification saves us from such sloppy thinking, but at the expense of having to make explicit the dependence of metavariables on bindable names via the use of function variables and application. For example, (1) would be replaced by something like

$$\frac{\mathtt{app}\,(\mathtt{fn}\,\lambda a.F\,a)\,X\ \Downarrow\ V}{\mathtt{let}\,X\,(\lambda a.F\,a)\ \Downarrow\ V} \tag{4}$$

or, modulo η-equivalence

$$\frac{\mathtt{app}\,(\mathtt{fn}\,F)\,X\ \Downarrow\ V}{\mathtt{let}\,X\,F\ \Downarrow\ V}\ . \tag{5}$$

Now goal (3) becomes $\mathtt{let}\,1\,\lambda b.b \Downarrow 1$ and there is no problem unifying it with the conclusion of (5) via a capture-avoiding substitution of 1 for X, $\lambda c.c$ for F and 1 for V.

This is all very fine, but the situation is not as pleasant as for first-order terms: higher-order unification problems can be undecidable, decidable but lack most general unifiers, or have such unifiers only by imposing some restrictions [3]; see [4] for a survey of higher-order unification. We started out wanting to compute with binders modulo α-equivalence, and somehow the process of making possibly-capturing substitution respectable has led to function variables, application, capture-avoiding substitution and $\beta\eta$-equivalence. Does it have to be so? No!

For one thing, several authors have already noted that one can make sense of possibly-capturing substitution modulo α-equivalence by using *explicit substitutions* in the term representation language: see [1,5–9]. Compared with those works, we make a number of simplifications. First, we find that we do not need to use function variables, application or $\beta\eta$-equivalence in our representation language—leaving just binders and α-equivalence. Secondly, instead of using explicit substitutions of names for names, we use only the special case of *explicit permutations* of names. The idea of using name-permutations, and in particular name-swappings, when dealing with α-conversion was described in [10] and there is growing evidence of its usefulness (see [11–13], for example). When a name substitution is actually a permutation, the function it induces from terms to terms is a bijection; this bijectivity gives the operation of permuting names very good logical properties compared with name substitution. Consider for example the α-equivalent terms $\mathtt{fn}\,a.b$ and $\mathtt{fn}\,c.b$, where a, b and c are distinct. If we apply the substitution $[b \mapsto a]$ (renaming all free occurrences of b to be a) to them we get $\mathtt{fn}\,a.a$ and $\mathtt{fn}\,c.a$, which are no longer α-equivalent. Thus renaming substitutions do not respect α-equivalence in general, and any unification algorithm using them needs to take extra precautions to not inadvertently change the intended meaning of terms. The traditional solution for this problem is to introduce a more complicated form of renaming substitution that avoids capture of names by binders. In contrast, the simple operation of name-permutation respects α-equivalence; for example, applying the name-permutation $(a\,b)$ that swaps all occurrences of a and b (be they free, bound or binding) to the terms above gives $\mathtt{fn}\,b.a$ and $\mathtt{fn}\,c.a$, which are still α-equivalent. We exploit such good properties of name-permutations to give a conceptually simple unification algorithm.

In addition to the use of explicit name-permutations, we also compute symbolically with predicates expressing *freshness* of names for terms. Such predicates certainly feature in previous work on binding (for example, Qu-Prolog's `not_free_in` predicate [8], the notion of "algebraic independence" in [14, Definition 3], and the "non-occurrence" predicates of [15]). But once again, the use of such a freshness predicate based upon name *swapping* rather than renaming, which arises naturally from the work reported in [10,16], gives us a simpler theory with good algorithmic properties. It is easy to see why there is a need for computing with freshness, given that we take a "nominal" approach to binders. (In other words we use concrete versions of binding and α-equivalence in which bound entities are named explicitly, rather than using de Bruijn-style representations, as for example in [1,7].) A basic instance of our generalised form of α-equivalence identifies $\mathtt{fn}\, a.X$ with $\mathtt{fn}\, b.(a\, b)\cdot X$ provided b *is fresh for* X, where the subterm $(a\, b)\cdot X$ indicates an explicit permutation—namely the swapping of a and b—waiting to be applied to X. We write "b is fresh for X" symbolically as $b \;\#\; X$; the intended meaning of this relation is that b does not occur free in any (ground) term that may be substituted for X. If we know more about X we may be able to eliminate the explicit permutation in $(a\, b)\cdot X$; for example, if we knew that $a \;\#\; X$ holds as well as $b \;\#\; X$, then $(a\, b)\cdot X$ can be replaced by X.

It should already be clear from these simple examples that in our setting the appropriate notion of term-equality is not a bare equation, $t \approx t'$, but rather a hypothetical judgement of the form

$$\nabla \vdash t \approx t' \tag{6}$$

where ∇ is a *freshness environment*—a finite set $\{a_1 \;\#\; X_1, \dots, a_n \;\#\; X_n\}$ of freshness assumptions. For example

$$\{a \;\#\; X, b \;\#\; X\} \vdash \mathtt{fn}\, a.X \approx \mathtt{fn}\, b.X \tag{7}$$

is a valid judgement of our *nominal equational logic*. Similarly, judgements about freshness itself will take the form

$$\nabla \vdash a \;\#\; t\ . \tag{8}$$

Two examples of valid freshness judgements are $\{a \;\#\; X\} \vdash a \;\#\; \mathtt{fn}\, b.X$ and $\emptyset \vdash a \;\#\; \mathtt{fn}\, a.X$.

The freshness environment ∇ in judgements of the form (6) and (8) expresses freshness conditions that any textual substitution of terms for variables must respect in order for the right-hand side of the judgement to be valid after substitution. This explicit use of freshness makes the operation of textual substitution respect our generalised form of α-equivalence. For example, if we were naïvely to regard the terms $\mathtt{fn}\, a.X$ and $\mathtt{fn}\, b.X$ as α-equivalent, then applying for example the capturing substitution $[X := a]$ or $[X := b]$ results into two terms that are *not* α-equivalent anymore. (A similar observation partly motivates the work in [17].) However, if we assume $a \# X$ and $b \# X$ as in (7), then all problematic substitutions are ruled out. In this way we obtain a version of α-equivalence between terms with variables that is respected by textual substitutions (see Lemma 2.14 below), unlike the traditional notion of α-equivalence.

Summary

We will represent languages involving binders using the usual notion of first-order terms over a many-sorted signature, but with certain distinguished constants and function symbols. These give us terms with: distinguished constants naming bindable entities, that we call *atoms*; terms $a.t$ expressing a generic form of *binding* of an atom a in a term t; and terms $\pi{\cdot}X$ representing an explicit *permutation* π of atoms waiting to be applied to whatever term is substituted for the variable X. Section 2 presents this term-language together with a syntax-directed inductive definition of the provable judgements of the form (6) and (8) which for *ground* terms (i.e. ones with no variables) agrees with the usual notions of α-equivalence and "not a free variable of". However, on open terms our judgements differ from these standard notions. Section 3 considers unification in this setting. Solving equalities between abstractions $a.t \approx ?\ a'.t'$ entails solving both equalities $t \approx ?\ (a\, a'){\cdot}t'$ and freshness problems $a\ \#?\ t'$. Therefore our general form of *nominal unification problem* is a finite collection of individual equality and freshness problems. Such a problem P is solved by providing not only a substitution σ (of terms for variables), but also a freshness environment ∇ (as above), which together have the property that $\nabla \vdash \sigma(t) \approx \sigma(t')$ and $\nabla \vdash a \# \sigma(t'')$ hold for each individual equality $t \approx ?\ t'$ and freshness $a\ \#?\ t''$ in the problem P. Our main result with respect to unification is that *solvability is decidable and that solvable problems possess most general solutions* (for a reasonably obvious notion of "most general"). The proof is via a unification algorithm that is very similar to the first-order algorithm given in the now-common transformational style [18]. (See [19, Sect. 2.6] or [20, Sect. 4.6] for expositions of this.) Section 4 considers the relationship of our version of "unification modulo α-equivalence" to existing approaches. Section 5 assesses what has been achieved and the prospects for applications.

Quiz

To appreciate the kind of problem that nominal unification solves, you might like to try the following quiz about the λ-calculus [21] before we apply our algorithm to solve it at the end of Section 3.

Assuming a and b are distinct variables, is it possible to find λ-terms $M_1, \ldots, M_7$ that make the following pairs of terms α-equivalent?

(1) $\lambda a.\lambda b.(M_1\, b)$ and $\lambda b.\lambda a.(a\, M_1)$
(2) $\lambda a.\lambda b.(M_2\, b)$ and $\lambda b.\lambda a.(a\, M_3)$
(3) $\lambda a.\lambda b.(b\, M_4)$ and $\lambda b.\lambda a.(a\, M_5)$
(4) $\lambda a.\lambda b.(b\, M_6)$ and $\lambda a.\lambda a.(a\, M_7)$

If it is possible to find a solution for any of these four problems, can you describe what all possible solutions for that problem are like? (The answers are given in Example 3.8.)

2 Nominal equational logic

We take a concrete approach to the syntax of binders in which bound entities are explicitly named. Furthermore we do not assume that the names of bound entities are necessarily variables (things that may be substituted for), in order to encompass examples like the π-calculus [22], in which the restriction operator binds channel names and these are quite different from names of unknown processes. Names of bound entities will be called *atoms*. This is partly for historical reasons (stemming from the work by the second two authors [10]) and partly to indicate that the internal structure of such names is irrelevant to us: all we care about is their identity (i.e. whether or not one atom is the same as another) and that the supply of atoms is inexhaustible.

Although there are several general frameworks in the literature for specifying languages with binders, not all of them meet the requirements mentioned in the previous paragraph. Use of the simply typed λ-calculus for this purpose is common; but as discussed in the Introduction, it leads to a problematic unification theory. Among *first-order* frameworks, Plotkin's notion of *binding signature* [23,24], being unsorted, equates names used in binding with names of variables standing for unknown terms; so it is not sufficiently general for us. A first-order framework that does meet our requirements is the notion of *nominal algebras* in [15]. The *nominal signatures* that we use in this paper are a mild (but practically useful) generalisation of nominal algebras in which name-abstraction and pairing can be mixed freely in arities (rather than insisting as in [15] that the argument sort of a function symbol be normalised to a tuple of abstractions).

Definition 2.1. A *nominal signature* is specified by: a set of *sorts of atoms* (typical symbol ν); a disjoint set of *sorts of data* (typical symbol δ); and a set of *function symbols* (typical symbol f), each of which has an *arity* of the form $\tau \to \delta$. Here τ ranges over (compound) *sorts* given by the grammar $\tau ::= \nu \mid \delta \mid 1 \mid \tau \times \tau \mid \langle\nu\rangle\tau$. Sorts of the form $\langle\nu\rangle\tau$ classify terms that are binding abstractions of atoms of sort ν over terms of sort τ. We will explain the syntax and properties of such terms in a moment.

Example 2.2. Here is a nominal signature for expressions in a small fragment of ML [25]:

sort of atoms:	vid
sort of data:	exp
function symbols:	$\mathtt{vr} : \mathit{vid} \rightarrow \mathit{exp}$
	$\mathtt{app} : \mathit{exp} \times \mathit{exp} \rightarrow \mathit{exp}$
	$\mathtt{fn} : \langle \mathit{vid} \rangle \mathit{exp} \rightarrow \mathit{exp}$
	$\mathtt{lv} : \mathit{exp} \times \langle \mathit{vid} \rangle \mathit{exp} \rightarrow \mathit{exp}$
	$\mathtt{lf} : \langle \mathit{vid} \rangle ((\langle \mathit{vid} \rangle \mathit{exp}) \times \mathit{exp}) \rightarrow \mathit{exp}$.

The function symbol `vr` constructs terms of sort exp representing value identifiers (named by atoms of sort vid); `app` constructs application expressions from pairs of expressions; `fn`, `lv` and `lf` construct terms representing respectively function abstractions (`fn x => e`), local value declarations (`let val x = e1 in e2 end`) and local recursive function declarations (`let fun f x = e1 in e2 end`). The arities of the function symbols specify which are binders and in which way their arguments are bound. For example, in the expression (`let fun f x = e1 in e2 end`) there is a binding occurrence of the value identifier `f` whose scope is both of `e1` and `e2`; and a binding occurrence of the value identifier `x` whose scope is just `e1`. These binding scopes are reflected by the argument sort of the function symbol `lf`. This kind of specification of binding scopes is of course a feature of *higher-order abstract syntax* [26], using function types $\nu{\rightarrow}\tau$ in simply typed λ-calculus where we use abstraction sorts $\langle\nu\rangle\tau$. We shall see that the latter have much more elementary (indeed, first-order) properties compared with the former.

Definition 2.3. Given a nominal signature, we assume that there are countably infinite and pairwise disjoint sets of *atoms* (typical symbol a) for each sort of atoms ν, and *variables* (typical symbol X) for each sort of atoms ν and each sort of data δ. The *terms* over a nominal signature and their sorts are inductively defined as follows, where we write $t : \tau$ to indicate that a term t has sort τ.

Unit value $\langle\rangle : 1$.
Pairs $\langle t_1, t_2\rangle : \tau_1 \times \tau_2$, if $t_1 : \tau_1$ and $t_2 : \tau_2$.
Data $f\,t : \delta$, if f is a function symbol of arity $\tau \rightarrow \delta$ and $t : \tau$.
Atoms $a : \nu$, if a is an atom of sort ν.
Atom-abstraction $a.t : \langle\nu\rangle\tau$, if a is an atom of sort ν and $t : \tau$.
Suspension $\pi{\cdot}X : \tau$, if $\pi = (a_1\,b_1)(a_2\,b_2)\cdots(a_n\,b_n)$ is a finite list whose elements $(a_i\,b_i)$ are pairs of atoms, with a_i and b_i of the same sort, and X is a variable of sort τ, where τ is either a sort of data or a sort of atoms (i.e. $\tau ::= \nu \mid \delta$).

Recall that every finite permutation can be expressed as a composition of swappings $(a_i\,b_i)$; the list π of pairs of atoms occurring in a suspension term $\pi{\cdot}X$ specifies a finite permutation of atoms waiting to be applied once we know more about the variable X (by substituting for it, for example). We represent finite permutations in this way because it is really the operation of swapping which plays a fundamental rôle in the theory. Since, semantically speaking, swapping commutes with all term-forming operations, we can normalise terms involving an explicit swapping operation by pushing the swap in as far as it will go, until it reaches a variable (applying the swapping to atoms that it meets on the way); the terms in Definition 2.3 are all normalised in this way, with explicit permutations "piled up" in front of variables giving what we have called *suspensions*. In case the permutation π in a suspension is the empty list, we just write X for $\pi{\cdot}X$.

Definition 2.4. The *permutation action*, $\pi{\cdot}t$, of a finite permutation of atoms π on a term t is defined as in Figure 1, making use of the following notations. The composition of a permutation π followed by a swap $(a\,b)$ is given by list-cons, written $(a\,b) :: \pi$. (Note that we apply permutations to terms on the left, and hence the order of the composition is from right to left.) The composition of π followed by another permutation π' is given by list-concatenation, written as $\pi' @ \pi$. The *identity* permutation is given by the empty list $[]$; and the *inverse* of a permutation is given by list reversal, written as π^{-1}.

Permutation actions have excellent logical properties (stemming from the fact that they are bijections). We exploit these properties in our definition of α-equivalence for terms over a nominal signature, which is respected by substitution of terms for variables even though the latter may involve capture of atoms by binders. To do so we will need to make use of an auxiliary relation of *freshness* between atoms and terms, whose intended meaning is that the atom does not occur free in any substitution instance of the term. As discussed in the Introduction, our judgements about term equivalence ($t \approx t'$) need to contain hypotheses about the freshness of atoms with respect to variables ($a \,\#\, X$); and the same goes for our judgements about freshness itself ($a \,\#\, t$). Figure 2 gives a syntax-directed inductive definition of equivalence and freshness using judgements of the form

$$\nabla \vdash t \approx t' \qquad \text{and} \qquad \nabla \vdash a \,\#\, t$$

where t and t' are terms of the same sort over a given nominal signature, a is an atom, and the *freshness environment* ∇ is a finite set of *freshness constraints* $a \,\#\, X$, each specified by an atom and a variable. Rule ($\approx$-suspension) in Figure 2 makes use of the following definition.

Definition 2.5. The *disagreement set* of two permutations π and π' is the set of atoms $ds(\pi, \pi') \stackrel{\text{def}}{=} \{a \mid \pi{\cdot}a \neq \pi'{\cdot}a\}$.

Note that every disagreement set $ds(\pi, \pi')$ is a subset of the *finite* set of atoms occurring in either of the lists π and π', because if a does not occur in those lists, then from Figure 1 we get $\pi{\cdot}a = a = \pi'{\cdot}a$. To illustrate the use of disagreement sets, consider the judgement

$$\{a \,\#\, X, c \,\#\, X\} \vdash (a\,c)(a\,b){\cdot}X \approx (b\,c){\cdot}X \ \ .$$

$$
\begin{aligned}
[\,]{\cdot}a &\stackrel{\text{def}}{=} a \\
((a_1\,a_2) :: \pi){\cdot}a &\stackrel{\text{def}}{=} \begin{cases} a_1 & \text{if } \pi{\cdot}a = a_2 \\ a_2 & \text{if } \pi{\cdot}a = a_1 \\ \pi{\cdot}a & \text{otherwise} \end{cases}
\end{aligned}
\qquad
\begin{aligned}
\pi{\cdot}\langle\rangle &\stackrel{\text{def}}{=} \langle\rangle \\
\pi{\cdot}\langle t_1, t_2\rangle &\stackrel{\text{def}}{=} \langle \pi{\cdot}t_1, \pi{\cdot}t_2\rangle \\
\pi{\cdot}(f\,t) &\stackrel{\text{def}}{=} f\,(\pi{\cdot}t) \\
\pi{\cdot}(a.t) &\stackrel{\text{def}}{=} (\pi{\cdot}a).(\pi{\cdot}t) \\
\pi{\cdot}(\pi'{\cdot}X) &\stackrel{\text{def}}{=} (\pi@\pi'){\cdot}X
\end{aligned}
$$

Fig. 1. Permutation action on terms, $\pi{\cdot}t$.

$$
\frac{}{\nabla \vdash \langle\rangle \approx \langle\rangle}\,(\approx\text{-unit})
\qquad
\frac{\nabla \vdash t_1 \approx t_1' \quad \nabla \vdash t_2 \approx t_2'}{\nabla \vdash \langle t_1, t_2\rangle \approx \langle t_1', t_2'\rangle}\,(\approx\text{-pair})
$$

$$
\frac{\nabla \vdash t \approx t'}{\nabla \vdash f\,t \approx f\,t'}\,(\approx\text{-function symbol})
\qquad
\frac{\nabla \vdash t \approx t'}{\nabla \vdash a.t \approx a.t'}\,(\approx\text{-abstraction-1})
$$

$$
\frac{a \neq a' \quad \nabla \vdash t \approx (a\,a'){\cdot}t' \quad \nabla \vdash a \# t'}{\nabla \vdash a.t \approx a'.t'}\,(\approx\text{-abstraction-2})
$$

$$
\frac{}{\nabla \vdash a \approx a}\,(\approx\text{-atom})
\qquad
\frac{(a \# X) \in \nabla \ \text{ for all } \ a \in ds(\pi, \pi')}{\nabla \vdash \pi{\cdot}X \approx \pi'{\cdot}X}\,(\approx\text{-suspension})
$$

$$
\frac{}{\nabla \vdash a \# \langle\rangle}\,(\#\text{-unit})
\qquad
\frac{\nabla \vdash a \# t_1 \quad \nabla \vdash a \# t_2}{\nabla \vdash a \# \langle t_1, t_2\rangle}\,(\#\text{-pair})
$$

$$
\frac{\nabla \vdash a \# t}{\nabla \vdash a \# f\,t}\,(\#\text{-function symbol})
$$

$$
\frac{}{\nabla \vdash a \# a.t}\,(\#\text{-abstraction-1})
\qquad
\frac{a \neq a' \quad \nabla \vdash a \# t}{\nabla \vdash a \# a'.t}\,(\#\text{-abstraction-2})
$$

$$
\frac{a \neq a'}{\nabla \vdash a \# a'}\,(\#\text{-atom})
\qquad
\frac{(\pi^{-1}{\cdot}a \# X) \in \nabla}{\nabla \vdash a \# \pi{\cdot}X}\,(\#\text{-suspension})
$$

Fig. 2. Inductive definition of $\approx$ and $\#$.

This holds by applying rule ($\approx$-suspension) in Figure 2, since the disagreement set of the permutations $(a\,c)(a\,b)$ and $(b\,c)$ is $\{a, c\}$.

Remark 2.6 (Freshness environments). Note that the freshness environment on the left-hand side of judgements in the rules in Figure 2 does not change from hypotheses to conclusion. So in the same way that we assume variables have attached sorting information, we could dispense with the use of freshness environments entirely by attaching the freshness information directly to variables. However, we find the use of freshness environments more elegant (for one thing, without them two variables with the same name but different freshness information would have to be regarded as different). They also make life simpler when we come on to nominal unification problems and their solutions in the next section.

Below we sketch a proof that $\approx$ is an equivalence relation. At first sight this property might be surprising considering the "unsymmetric" definition of the rule ($\approx$-abstraction-2). However it holds because of the good logical properties of the relation $\approx$ with respect to permutation actions. Although reasoning about $\approx$ is rather pleasant once equivalence is proved, establishing it first is rather tricky—mainly because of the large number of cases, but also because several facts needed in the proof are interdependent. [1] We first show that permutations can be moved from one side of the freshness relation to the other by forming the inverse permutation, and that the freshness relation is preserved under permutation actions.

Lemma 2.7.

(1) If $\nabla \vdash a \# \pi{\cdot}t$ *then* $\nabla \vdash \pi^{-1}{\cdot}a \# t$.
(2) If $\nabla \vdash \pi{\cdot}a \# t$ *then* $\nabla \vdash a \# \pi^{-1}{\cdot}t$.
(3) If $\nabla \vdash a \# t$ *then* $\nabla \vdash \pi{\cdot}a \# \pi{\cdot}t$.

PROOF. (1) and (2) are by routine inductions on the structure of t, using the fact that $\pi{\cdot}a = b$ iff $a = \pi^{-1}{\cdot}b$; (3) is a consequence of (2) and the fact that permutations are bijections on atoms. □

According to the definition of the permutation action given in Figure 1, if we push a permutation inside a term, we need to apply the permutation to all atoms we meet on the way. Suppose we apply two distinct permutations, say π and π', to a term t, then in general $\pi{\cdot}t$ and $\pi'{\cdot}t$ are not α-equivalent—the disagreement set $ds(\pi, \pi')$ characterises all atoms which potentially lead to differences. However, if we assume that all atoms in $ds(\pi, \pi')$ are fresh for t, then we can infer that the permutation actions produce equivalent terms. This is made precise in the following lemma.

Lemma 2.8. *Given any* π *and* π'*, if* $\nabla \vdash a \# t$ *holds for all* $a \in ds(\pi, \pi')$*, then* $\nabla \vdash \pi{\cdot}t \approx \pi'{\cdot}t$*.*

PROOF. By induction on the structure of t, for all π and π' simultaneously, using the fact about disagreement sets that for all atoms a, b, if $a \in ds(\pi, (\pi{\cdot}b\ \ \pi'{\cdot}b) :: \pi')$ then $a \in ds(\pi, \pi')$. □

An example of this lemma is that $\nabla \vdash \pi{\cdot}(a\,b){\cdot}t \approx (\pi{\cdot}a\ \ \pi{\cdot}b){\cdot}\pi{\cdot}t$ is a valid judgement, because the disagreement set $ds(\pi@(a\,b)\,,\,(\pi{\cdot}a\ \ \pi{\cdot}b) :: \pi)$ is empty.

[1] In addition some further simple properties of permutations and disagreement sets need to be established first. A machine-checked proof of *all* results using the theorem prover Isabelle can be found at `http://www.cl.cam.ac.uk/users/cu200/Unification`.

The next lemma shows that $\approx$ respects the freshness relation.

Lemma 2.9. *If* $\nabla \vdash a \# t$ *and* $\nabla \vdash t \approx t'$*, then* $\nabla \vdash a \# t'$.

PROOF. Routine induction on the definition of $\approx$ using Lemma 2.7. $\square$

For showing transitivity of the relation $\approx$, it will be necessary to define a measure that counts all term constructors occurring in a term.

Definition 2.10. The *size* of a term t is the natural number $|t|$ defined by:

$$
\begin{aligned}
|\pi{\cdot}X|, |a|, |\langle\rangle| &\stackrel{\text{def}}{=} 1 \\
|a.t|, |f\, t| &\stackrel{\text{def}}{=} 1 + |t| \\
|\langle t_1, t_2 \rangle| &\stackrel{\text{def}}{=} 1 + |t_1| + |t_2|
\end{aligned}
$$

Notice that the size of a term is preserved under permutation actions (i.e. $|\pi{\cdot}t| = |t|$) and respected by the relation $\approx$ in the sense that if $\nabla \vdash t \approx t'$ then $|t| = |t'|$.

Theorem 2.11 (Equivalence). $\nabla \vdash - \approx -$ *is an equivalence relation.*

PROOF. Reflexivity is by a simple induction on the structure of terms. Transitivity is by an induction on the size of terms: a slight complication is that many subcases need to be analysed (for example five subcases when dealing with abstractions) and also that transitivity needs to be shown by mutual induction with the fact that $\approx$ is preserved under permutation actions, that is

$$\text{given any } \pi, \text{ if } \nabla \vdash t \approx t' \text{ then } \nabla \vdash \pi{\cdot}t \approx \pi{\cdot}t' \; . \tag{9}$$

We illustrate the proof of transitivity for the case when $\nabla \vdash a_1.t_1 \approx a_2.t_2$ and $\nabla \vdash a_2.t_2 \approx a_3.t_3$ hold, with a_1, a_2 and a_3 all distinct atoms, and we have to prove $\nabla \vdash a_1.t_1 \approx a_3.t_3$. By the ($\approx$-abstraction-2) rule we can infer from the assumptions the following facts:

$$
\begin{array}{llll}
\text{(i)} & \nabla \vdash t_1 \approx (a_1\, a_2){\cdot}t_2 & \text{(ii)} & \nabla \vdash a_1 \# t_2 \\
\text{(iii)} & \nabla \vdash t_2 \approx (a_2\, a_3){\cdot}t_3 & \text{(iv)} & \nabla \vdash a_2 \# t_3
\end{array}
$$

Below we give the steps that prove $\nabla \vdash a_1.t_1 \approx a_3.t_3$.

(a) $\nabla \vdash (a_1\, a_2){\cdot}t_2 \approx (a_1\, a_2)(a_2\, a_3){\cdot}t_3$ by (iii) and IH (9)

(b) $\nabla \vdash t_1 \approx (a_1\, a_2)(a_2\, a_3){\cdot}t_3$ by (i), (a) and IH (transitivity)

(c) $ds((a_1\, a_2)(a_2\, a_3)\, ,\, (a_1\, a_3)) = \{a_1, a_2\}$ by definition

(d) $\nabla \vdash a_1 \# (a_2\, a_3){\cdot}t_3$ by (ii), (iii) and Lemma 2.9

(e) $\nabla \vdash a_1 \# t_3$ by $(a_2\, a_3){\cdot}a_1 = a_1$, (d) and Lemma 2.7(i)

(f) $\nabla \vdash (a_1\, a_2)(a_2\, a_3){\cdot}t_3 \approx (a_1\, a_3){\cdot}t_3$ by (c), (iv), (e) and Lemma 2.8

(g) $\nabla \vdash t_1 \approx (a_1\, a_3){\cdot}t_3$ by (b), (f) and IH (transitivity)

(h) $\nabla \vdash a_1.t_1 \approx a_3.t_3$ by (e), (g) and ($\approx$-abstraction-2)

The other cases are by similar arguments. Symmetry is then by a routine induction on the definition of $\approx$ using Lemma 2.8 and transitivity. □

Now it is relatively straightforward to obtain the following properties of our equivalence relation with respect to permutation actions.

Corollary 2.12.

(1) $\nabla \vdash t \approx \pi^{-1}{\cdot}\pi{\cdot}t'$ *if and only if* $\nabla \vdash t \approx t'$.
(2) $\nabla \vdash t \approx \pi{\cdot}t'$ *if and only if* $\nabla \vdash \pi^{-1}{\cdot}t \approx t'$.
(3) Given any π *and* π'*, if* $\nabla \vdash \pi{\cdot}t \approx \pi'{\cdot}t$ *then for all* a *in* $ds(\pi, \pi')$ *we have* $\nabla \vdash a \# t$*.*

PROOF. (i) follows immediately from Lemma 2.8 and transitivity; (ii) follows from (9) and (i); and (iii) is by a routine induction on the structure of t using Lemma 2.9. □

The main reason for using suspensions in the syntax of terms is to enable a definition of *substitution of terms for variables* that allows capture of free atoms by atom-abstractions while still respecting α-equivalence. The following lemma establishes this. First we give some terminology and notation for term-substitution.

Definition 2.13. A *substitution* σ is a sort-respecting function from variables to terms with the property that $\sigma(X) = X$ for all but finitely many variables X. We write $dom(\sigma)$ for the finite set of variables X satisfying $\sigma(X) \neq X$. If $dom(\sigma)$ consists of distinct variables $X_1, \ldots, X_n$ and $\sigma(X_i) = t_i$ for $i = 1..n$, we sometimes write σ as

$$\sigma = [X_1 := t_1, \ldots, X_n := t_n]. \tag{10}$$

We write $\sigma(t)$ for the result of *applying a substitution* σ to a term t; this is the term obtained from t by replacing each suspension $\pi{\cdot}X$ in t (as X ranges over

$dom(\sigma)$) by the term $\pi{\cdot}\sigma(X)$ got by letting π act on the term $\sigma(X)$ using the definition in Figure 1. For example, if $\sigma = [X := \langle b, Y\rangle]$ and $t = a.(a\,b){\cdot}X$, then $\sigma(t) = a.\langle a, (a\,b){\cdot}Y\rangle$. Given substitutions σ and σ', and freshness environments ∇ and ∇', we write

$$\text{(a)} \quad \nabla' \vdash \sigma(\nabla) \qquad \text{and} \qquad \text{(b)} \quad \nabla \vdash \sigma \approx \sigma' \tag{11}$$

to mean, for (a), that $\nabla' \vdash a \,\#\, \sigma(X)$ holds for each $(a \,\#\, X) \in \nabla$ and, for (b), that $\nabla \vdash \sigma(X) \approx \sigma'(X)$ holds for all $X \in dom(\sigma) \cup dom(\sigma')$.

Lemma 2.14 (Substitution). *Substitution commutes with the permutation action: $\sigma(\pi{\cdot}t) = \pi{\cdot}(\sigma(t))$. Substitution also preserves $\approx$ and $\#$ in the following sense:*

(1) if $\nabla' \vdash \sigma(\nabla)$ and $\nabla \vdash t \approx t'$, then $\nabla' \vdash \sigma(t) \approx \sigma(t')$;
(2) if $\nabla' \vdash \sigma(\nabla)$ and $\nabla \vdash a \,\#\, t$, then $\nabla' \vdash a \,\#\, \sigma(t)$.

PROOF. The first sentence follows by a routine induction on the structure of t. The second follows by induction on the definition of $\approx$ and $\#$ using Lemma 2.8. □

We claim that the relation $\approx$ defined in Figure 2 gives the correct notion of α-equivalence for terms over a nominal signature. This is reasonable, given Theorem 2.11 and the fact that, by definition, it satisfies rules ($\approx$-abstraction-1) and ($\approx$-abstraction-2). Further evidence is provided by the following proposition, which shows that for ground terms $\approx$ agrees with the following more traditional definition of α-equivalence.

Definition 2.15 (Naïve α-equivalence). Define the binary relation $t =_\alpha t'$ between terms over a nominal signature to be the least sort-respecting congruence relation satisfying $a.t =_\alpha b.[a{\mapsto}b]t$ whenever b is an atom (of the same sort as a) not occurring at all in the term t. Here $[a{\mapsto}b]t$ indicates the result of replacing all free occurrences of a with b in t.

Proposition 2.16 (Adequacy). *If t and t' are ground terms* (i.e. terms with no variables and hence no suspensions) *over a nominal signature, then the relation $t =_\alpha t'$ of Definition 2.15 holds if and only if $\emptyset \vdash t \approx t'$ is provable from the rules in Figure 2. Furthermore, $\emptyset \vdash a \,\#\, t$ is provable if and only if a is not in the set*

$FA(t)$ of free atoms of t, defined by:

$$\begin{aligned}
FA(\langle\rangle) &\stackrel{\textit{def}}{=} \emptyset \\
FA(\langle t_1, t_2\rangle) &\stackrel{\textit{def}}{=} FA(t_1) \cup FA(t_2) \\
FA(f\,t) &\stackrel{\textit{def}}{=} FA(t) \\
FA(a) &\stackrel{\textit{def}}{=} \{a\} \\
FA(a.t) &\stackrel{\textit{def}}{=} FA(t) - \{a\}.
\end{aligned}$$

PROOF. The proof is similar to the proof of [10, Proposition 2.2]. □

For non-ground terms, the relations $=_\alpha$ and $\approx$ differ! For example $a.X =_\alpha b.X$ always holds, whereas $\emptyset \vdash a.X \approx b.X$ is not provable unless $a = b$. This disagreement is to be expected, since we noted in the Introduction that $=_\alpha$ is *not* preserved by substitution, whereas from Lemma 2.14 we know that $\approx$ is.

3 Unification

Given terms t and t' of the same sort over a nominal signature, can we decide whether or not there is a substitution of terms for the variables in t and t' that makes them equal in the sense of the relation $\approx$ introduced in the previous section? Since instances of $\approx$ are established modulo freshness constraints, it makes more sense to ask whether or not there is both a substitution σ and a freshness environment ∇ for which $\nabla \vdash \sigma(t) \approx \sigma(t')$ holds. As for ordinary first-order unification, solving such an equational problem may throw up *several* equational subproblems; but an added complication here is that because of rule ($\approx$-abstraction-2) in Figure 2, equational problems may generate *freshness* problems, i.e. ones involving the relation $\#$. We are thus led to the following definition of unification problems for nominal equational logic.

Definition 3.1. A *unification problem* P over a nominal signature is a finite set of atomic problems, each of which is either an *equational problem* $t \approx ?\ t'$ where t and t' are terms of the same sort over the signature, or a *freshness problem* $a \mathrel{\#?}\ t$ where a is an atom and t a term over the signature. A *solution* for P consists of a pair (∇, σ) where ∇ is a freshness environment and σ is a substitution satisfying

- $\nabla \vdash a \mathrel{\#} \sigma(t)$ for each $(a \mathrel{\#?}\ t) \in P$ and
- $\nabla \vdash \sigma(t) \approx \sigma(t')$ for each $(t \approx ?\ t') \in P$.

We write $\mathcal{U}(P)$ for the set of all solutions of a problem P. $(\nabla, \sigma) \in \mathcal{U}(P)$ is a *most general* solution for P if given any other solution $(\nabla', \sigma') \in \mathcal{U}(P)$, then there is a substitution σ'' satisfying $\nabla' \vdash \sigma''(\nabla)$ and $\nabla' \vdash \sigma'' \circ \sigma \approx \sigma'$. (Here we are using the notation of (11); and $\sigma'' \circ \sigma$ denotes the *substitution composition* of σ followed by σ'', given by $(\sigma'' \circ \sigma)(X) \stackrel{\text{def}}{=} \sigma''(\sigma(X))$.) A solution $(\nabla, \sigma) \in \mathcal{U}(P)$ is *idempotent* provided $\nabla \vdash \sigma \circ \sigma \approx \sigma$.

We describe an algorithm which, given any nominal unification problem, decides whether or not it has a solution and if it does, returns a most general (and idempotent) solution. The algorithm uses labelled transformations, directly generalising the presentation of first-order unification in [19, Sect. 2.6] which in turn is based upon the approach in [18]. (See also [20, Sect. 4.6] for a detailed exposition, but not using labels.) We use two types of labelled transformation between unification problems, namely

$$P \stackrel{\sigma}{\Longrightarrow} P' \quad \text{and} \quad P \stackrel{\nabla}{\Longrightarrow} P'$$

where the substitution σ is either the identity ε, or a single replacement $[X := t]$; and where the freshness environment ∇ is either empty $\emptyset$, or a singleton $\{a \# X\}$. The legal transformations are given in Figure 3. This figure uses the notation $P \uplus P'$ to indicate the union of problems P and P' that are disjoint ($P \cap P' = \emptyset$); and the notation σP to indicate the problem resulting from applying the substitution σ to all the terms occurring in the problem P.

Algorithm. Given a unification problem P, the algorithm proceeds in two phases. [2] In the first phase it applies as many $\stackrel{\sigma}{\Longrightarrow}$ transformations as possible (non-deterministically). If this results in a problem containing no equational subproblems, then it proceeds to the second phase; otherwise it halts signalling failure. In the second phase it applies as many $\stackrel{\nabla}{\Longrightarrow}$ transformations as possible (non-deterministically). If this does not result in the empty problem, then it halts signalling failure; otherwise overall it has constructed a transformation sequence of the form

$$P \stackrel{\sigma_1}{\Longrightarrow} \cdots \stackrel{\sigma_n}{\Longrightarrow} P' \stackrel{\nabla_1}{\Longrightarrow} \cdots \stackrel{\nabla_m}{\Longrightarrow} \emptyset \tag{12}$$

(where P' does not contain any equational subproblems) and the algorithm returns the solution $(\nabla_1 \cup \cdots \cup \nabla_m,\ \sigma_n \circ \cdots \circ \sigma_1)$.

To show the correctness of this algorithm, we first establish that all sequences of unification transitions must terminate.

Lemma 3.2. *There is no infinite series of unification transitions.*

PROOF. Since every reduction sequence consists of two (possibly empty) subsequences, namely one containing only $\stackrel{\sigma}{\Longrightarrow}$-steps and the other only $\stackrel{\nabla}{\Longrightarrow}$-steps, we can show termination for both subsequences separately. For every unification problem P we define a measure of the size of P to be the lexicographically ordered pair

[2] See Remark 3.9 for discussion of this use of two phases.

$$
\begin{array}{lrcll}
(\approx?\text{-unit}) & \{\langle\rangle \approx? \langle\rangle\} \uplus P & \stackrel{\varepsilon}{\Longrightarrow} & P & \\
(\approx?\text{-pair}) & \{\langle t_1, t_2\rangle \approx? \langle t'_1, t'_2\rangle\} \uplus P & \stackrel{\varepsilon}{\Longrightarrow} & \{t_1 \approx? t'_1, t_2 \approx? t'_2\} \cup P & \\
(\approx?\text{-function symbol}) & \{f\,t \approx? f\,t'\} \uplus P & \stackrel{\varepsilon}{\Longrightarrow} & \{t \approx? t'\} \cup P & \\
(\approx?\text{-abstraction-1}) & \{a.t \approx? a.t'\} \uplus P & \stackrel{\varepsilon}{\Longrightarrow} & \{t \approx? t'\} \cup P & \\
(\approx?\text{-abstraction-2}) & \{a.t \approx? a'.t'\} \uplus P & \stackrel{\varepsilon}{\Longrightarrow} & \{t \approx? (a\,a')\boldsymbol{\cdot}t', a \#? t'\} \cup P & \\
 & & & & \text{provided } a \neq a' \\
(\approx?\text{-atom}) & \{a \approx? a\} \uplus P & \stackrel{\varepsilon}{\Longrightarrow} & P & \\
(\approx?\text{-suspension}) & \{\pi\boldsymbol{\cdot}X \approx? \pi'\boldsymbol{\cdot}X\} \uplus P & \stackrel{\varepsilon}{\Longrightarrow} & \{a \#? X \mid a \in ds(\pi, \pi')\} \cup P & \\
(\approx?\text{-variable}) & \left.\begin{array}{r}\{t \approx? \pi\boldsymbol{\cdot}X\} \uplus P \\ \{\pi\boldsymbol{\cdot}X \approx? t\} \uplus P\end{array}\right\} & \stackrel{\sigma}{\Longrightarrow} & \sigma P \quad \text{with } \sigma = [X := \pi^{-1}\boldsymbol{\cdot}t] & \\
 & & & & \text{provided } X \text{ does not occur in } t \\
(\#?\text{-unit}) & \{a \#? \langle\rangle\} \uplus P & \stackrel{\emptyset}{\Longrightarrow} & P & \\
(\#?\text{-pair}) & \{a \#? \langle t_1, t_2\rangle\} \uplus P & \stackrel{\emptyset}{\Longrightarrow} & \{a \#? t_1, a \#? t_2\} \cup P & \\
(\#?\text{-function symbol}) & \{a \#? f\,t\} \uplus P & \stackrel{\emptyset}{\Longrightarrow} & \{a \#? t\} \cup P & \\
(\#?\text{-abstraction-1}) & \{a \#? a.t\} \uplus P & \stackrel{\emptyset}{\Longrightarrow} & P & \\
(\#?\text{-abstraction-2}) & \{a \#? a'.t\} \uplus P & \stackrel{\emptyset}{\Longrightarrow} & \{a \#? t\} \cup P & \text{provided } a \neq a' \\
(\#?\text{-atom}) & \{a \#? a'\} \uplus P & \stackrel{\emptyset}{\Longrightarrow} & P & \text{provided } a \neq a' \\
(\#?\text{-suspension}) & \{a \#? \pi\boldsymbol{\cdot}X\} \uplus P & \stackrel{\nabla}{\Longrightarrow} & P \quad \text{with } \nabla = \{\pi^{-1}\boldsymbol{\cdot}a \# X\} &
\end{array}
$$

Fig. 3. Labelled transformations.

of natural numbers (n_1, n_2), where n_1 is the number of different variables used in P, and n_2 is the size (see Definition 2.10) of all equational problems in P, that is

$$n_2 \stackrel{\text{def}}{=} \sum_{(t \approx? t') \in P} |t| + |t'| \;.$$

In every $\stackrel{\sigma}{\Longrightarrow}$-step this measure decreases: the ($\approx?$-variable) transition eliminates (completely) one variable from the unification problem, and therefore n_1 decreases; the ($\approx?$-suspension) transition may eliminate a variable and also decreases the size n_2; all other transitions leave the number of variables unchanged, but decrease n_2. For the $\stackrel{\nabla}{\Longrightarrow}$-steps the size

$$\sum_{(a \#? t) \in P} |t|$$

decreases in every step. Taking both facts together means that every reduction sequence must terminate. □

The following lemmas help us to show that the algorithm gives correct results upon termination.

Lemma 3.3. *If* $\nabla \vdash \sigma(\pi{\cdot}X) \approx \sigma(t)$ *then* $\nabla \vdash \sigma \circ [X := \pi^{-1}{\cdot}t] \approx \sigma$.

PROOF. We have to prove that both substitutions agree (modulo $\approx$) on all variables in $dom(\sigma) \cup \{X\}$. The only interesting case is for the substitutions applied to X, when we need to show that $\nabla \vdash \sigma(\pi^{-1}{\cdot}t) \approx \sigma(X)$. By Lemma 2.14 we can commute the permutation to the outside and move it to the other side of $\approx$ by Lemma 2.12—this gives $\nabla \vdash \sigma(t) \approx \pi{\cdot}\sigma(X)$. The case then follows from the assumptions by symmetry and commuting the permutation inside the substitution. $\square$

Lemma 3.4. *Given a unification problem* P, $(\nabla, \sigma) \in \mathcal{U}(\sigma' P)$ *holds if and only if* $(\nabla, \sigma \circ \sigma') \in \mathcal{U}(P)$.

PROOF. Simple calculation using the fact that $\sigma(\sigma'(t)) = (\sigma \circ \sigma')(t)$. $\square$

The following two lemmas show that the unification transformations can be used to determine whether or not solutions exists and to describe all of them if they do exist.

Lemma 3.5.

(i) *If* $(\nabla', \sigma') \in \mathcal{U}(P)$ *and* $P \overset{\sigma}{\Longrightarrow} P'$, *then* $(\nabla', \sigma') \in \mathcal{U}(P')$ *and* $\nabla' \vdash \sigma' \circ \sigma \approx \sigma'$.
(ii) *If* $(\nabla', \sigma') \in \mathcal{U}(P)$ *and* $P \overset{\nabla}{\Longrightarrow} P'$, *then* $(\nabla', \sigma') \in \mathcal{U}(P')$ *and* $\nabla' \vdash \sigma'(\nabla)$.

PROOF. We just give the details for two unification transitions: the case for ($\approx$?-suspension) follows from Lemma 2.12(iii); and the ($\approx$?-variable) case is a consequence of Lemmas 3.3 and 3.4. $\square$

Lemma 3.6.

(i) *If* $(\nabla', \sigma') \in \mathcal{U}(P')$ *and* $P \overset{\sigma}{\Longrightarrow} P'$, *then* $(\nabla', \sigma' \circ \sigma) \in \mathcal{U}(P)$.
(ii) *If* $(\nabla', \sigma') \in \mathcal{U}(P')$, $P \overset{\nabla}{\Longrightarrow} P'$ *and* $\nabla'' \vdash \sigma'(\nabla)$, *then* $(\nabla' \cup \nabla'', \sigma') \in \mathcal{U}(P)$.

PROOF. Once again, we just give the details for two unification transitions: the ($\approx$?-suspension) case follows from Lemma 2.8; and the ($\approx$?-variable) case follows from Lemma 3.4 and the fact that $t[X := \pi^{-1}{\cdot}t] = t$, which holds by the side-condition on the ($\approx$?-variable) transition about the non-occurrence of X in t. $\square$

The following theorem establishes the correctness of the nominal unification algorithm and is the central result of the paper.

Theorem 3.7 (Correctness). *Given a unification problem P*

(i) if the algorithm fails on P, then P has no solution; and
(ii) if the algorithm succeeds on P, then the result it produces is an idempotent most general solution.

PROOF. When failure happens it is because of certain subproblems that manifestly have no solution (namely in the first phase, $a \approx ?\ a'$ with $a \neq a'$, and $\pi{\boldsymbol\cdot}X \approx ?\ f\,t$ or $f\,t \approx ?\ \pi{\boldsymbol\cdot}X$ with X occurring in t; in the second phase, $a \ \#?\ a$). Therefore part (i) is a consequence of Lemma 3.5. For part (ii) one gets that a sequence like (12) exists, and thus $(\nabla, \sigma) = (\nabla_1 \cup \cdots \cup \nabla_m,\ \sigma_n \circ \cdots \circ \sigma_1)$ is in $\mathcal{U}(P)$ by Lemma 3.6 and the fact that $(\emptyset, \varepsilon) \in \mathcal{U}(\emptyset)$. Furthermore from Lemma 3.5, we get that any other solution $(\nabla', \sigma') \in \mathcal{U}(P)$ satisfies $\nabla' \vdash \sigma'(\nabla)$ and $\nabla' \vdash \sigma' \circ \sigma \approx \sigma'$, so that (∇, σ) is indeed a most general solution. Since one of those solutions is the most general solution (∇, σ), we also know that $\nabla \vdash \sigma \circ \sigma \approx \sigma$ and hence that (∇, σ) is idempotent. □

We now apply the nominal unification algorithm to solve the quiz questions from the Introduction.

Example 3.8. Using the first three function symbols of the nominal signature of Example 2.2 to represent λ-terms, the Quiz at the end of the Introduction translates into the following four unification problems over that signature, where a and b are distinct atoms of sort vid and $X_1, \ldots, X_7$ are distinct variables of sort exp:

$$P_1 \stackrel{\text{def}}{=} \{\mathtt{fn}\,a.\mathtt{fn}\,b.\mathtt{app}\langle X_1, \mathtt{vr}\,b\rangle \approx ?\ \mathtt{fn}\,b.\mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_1\rangle\},$$
$$P_2 \stackrel{\text{def}}{=} \{\mathtt{fn}\,a.\mathtt{fn}\,b.\mathtt{app}\langle X_2, \mathtt{vr}\,b\rangle \approx ?\ \mathtt{fn}\,b.\mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_3\rangle\},$$
$$P_3 \stackrel{\text{def}}{=} \{\mathtt{fn}\,a.\mathtt{fn}\,b.\mathtt{app}\langle \mathtt{vr}\,b, X_4\rangle \approx ?\ \mathtt{fn}\,b.\mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_5\rangle\},$$
$$P_4 \stackrel{\text{def}}{=} \{\mathtt{fn}\,a.\mathtt{fn}\,b.\mathtt{app}\langle \mathtt{vr}\,b, X_6\rangle \approx ?\ \mathtt{fn}\,a.\mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_7\rangle\}.$$

Applying the nominal unification algorithm described above, we find that

- P_1 has no solution;
- P_2 has a most general solution given by $\nabla_2 = \emptyset$ and $\sigma_2 = [X_2 := \mathtt{vr}\,b, X_3 := \mathtt{vr}\,a]$;
- P_3 has a most general solution given by $\nabla_3 = \emptyset$ and $\sigma_3 = [X_4 := (a\,b){\boldsymbol\cdot}X_5]$;
- P_4 has a most general solution given by $\nabla_4 = \{b \ \#\ X_7\}$ and $\sigma_3 = [X_6 := (b\,a){\boldsymbol\cdot}X_7]$.

$$
\begin{array}{lll}
P_1 \overset{\varepsilon}{\Longrightarrow} \{\mathtt{fn}\, b.\mathtt{app}\langle X_1, \mathtt{vr}\, b\rangle \approx?\ \mathtt{fn}\, b.\mathtt{app}\langle \mathtt{vr}\, b, (a\, b){\cdot}X_1\rangle,\ a \#?\ \mathtt{fn}\, a.\mathtt{app}\langle \mathtt{vr}\, a, X_1\rangle\} & (\approx?\text{-abstraction-2})\\
\quad\overset{\varepsilon}{\Longrightarrow} \{\mathtt{app}\langle X_1, \mathtt{vr}\, b\rangle \approx?\ \mathtt{app}\langle \mathtt{vr}\, b, (a\, b){\cdot}X_1\rangle,\ a \#?\ \mathtt{fn}\, a.\mathtt{app}\langle \mathtt{vr}\, a, X_1\rangle\} & (\approx?\text{-abstraction-1})\\
\quad\ldots\ \ldots & \ldots\\
\quad\overset{\varepsilon}{\Longrightarrow} \{X_1 \approx?\ \mathtt{vr}\, b,\ \mathtt{vr}\, b \approx?\ (a\, b){\cdot}X_1,\ a \#?\ \mathtt{fn}\, a.\mathtt{app}\langle \mathtt{vr}\, a, X_1\rangle\} & (\approx?\text{-pair})\\
\quad\overset{\sigma}{\Longrightarrow} \{\mathtt{vr}\, b \approx?\ \mathtt{vr}\, a,\ a \#?\ \mathtt{fn}\, a.\mathtt{app}\langle \mathtt{vr}\, a, \mathtt{vr}\, b\rangle\}\ \text{ with } \sigma = [X_1 := \mathtt{vr}\, b] & (\approx?\text{-variable})\\
\quad\overset{\varepsilon}{\Longrightarrow} \{b \approx?\ a,\ a \#?\ \mathtt{fn}\, a.\mathtt{app}\langle \mathtt{vr}\, a, \mathtt{vr}\, b\rangle\} & (\approx?\text{-function symbol})\\
\quad\quad \text{FAIL} & \\
 & \\
P_4 \overset{\varepsilon}{\Longrightarrow} \{\mathtt{fn}\, b.\mathtt{app}\langle \mathtt{vr}\, b, X_6\rangle \approx?\ \mathtt{fn}\, a.\mathtt{app}\langle \mathtt{vr}\, a, X_7\rangle\} & (\approx?\text{-abstraction-1})\\
\quad\overset{\varepsilon}{\Longrightarrow} \{\mathtt{app}\langle \mathtt{vr}\, b, X_6\rangle \approx?\ \mathtt{app}\langle \mathtt{vr}\, b, (b\, a){\cdot}X_7\rangle,\ b \#?\ \mathtt{app}\langle \mathtt{vr}\, a, X_7\rangle\} & (\approx?\text{-abstraction-2})\\
\quad\ldots\ \ldots & \ldots\\
\quad\overset{\varepsilon}{\Longrightarrow} \{b \approx?\ b,\ X_6 \approx?\ (b\, a){\cdot}X_7,\ b \#?\ \mathtt{app}\langle \mathtt{vr}\, a, X_7\rangle\} & (\approx?\text{-function symbol})\\
\quad\overset{\varepsilon}{\Longrightarrow} \{X_6 \approx?\ (b\, a){\cdot}X_7,\ b \#?\ \mathtt{app}\langle \mathtt{vr}\, a, X_7\rangle\} & (\approx?\text{-atom})\\
\quad\overset{\sigma}{\Longrightarrow} \{b \#?\ \mathtt{app}\langle \mathtt{vr}\, a, X_7\rangle\}\quad \text{with } \sigma = [X_6 := (b\, a){\cdot}X_7] & (\approx?\text{-variable})\\
\quad\overset{\emptyset}{\Longrightarrow} \{b \#?\ \langle \mathtt{vr}\, a, X_7\rangle\} & (\#?\text{-function symbol})\\
\quad\ldots\ \ldots & \ldots\\
\quad\overset{\emptyset}{\Longrightarrow} \{b \#?\ a,\ b \#?\ X_7\} & (\#?\text{-function symbol})\\
\quad\overset{\emptyset}{\Longrightarrow} \{b \#?\ X_7\} & (\#?\text{-atom})\\
\quad\overset{\nabla}{\Longrightarrow} \emptyset \quad \text{with } \nabla = \{b \# X_7\} & (\#?\text{-suspension})
\end{array}
$$

Fig. 4. Example derivations

Derivations for P_1 and P_4 are sketched in Figure 4. Using the Adequacy property of Proposition 2.16, one can interpret these solutions as the following statements about the λ-terms mentioned in the quiz.

Quiz answers

(1) There is no λ-term M_1 making the first pair of terms α-equivalent.
(2) The only solution for the second problem is to take $M_2 = b$ and $M_3 = a$.
(3) For the third problem we can take M_5 to be any λ-term, so long as we take M_4 to be the result of swapping all occurrences of a and b throughout M_5.
(4) For the last problem, we can take M_7 to be any λ-term that *does not contain free occurrences of* b, so long as we take M_6 to be the result of swapping all occurrences of b and a throughout M_7, or equivalently (since b is not free in M_7), taking M_6 to be the result of replacing all free occurrences of a in M_7 with b.

Remark 3.9 (Separation of the algorithm into two phases). We organised the algorithm into two phases: equation-solving followed by freshness-solving. Note that the second phase is crucial for the soundness of the algorithm. Consider for example the unification problem consisting of two terms which are *not* α-equivalent:

$$\{a.b \approx?\ b.a\} \tag{13}$$

After applying the transformation ($\approx?$-abstraction-2) one needs to solve the problem $\{a \approx ?\ a, a \ \#?\ a\}$, whose first component is solved by ($\approx$?-atom). Failure is only signalled by the algorithm in the second phase when attempting to solve the unsolvable freshness problem $\{a \ \#?\ a\}$. The second phase, i. e. solving all freshness problems, ensures that the unifiers calculated by the algorithm are sound with respect to our notion of α-equivalence.

We used this separation of the algorithm into two phases in order to make the correctness proof easier. More efficient algorithms would seek to minimise the amount of redundant calculations before failures are signalled, by solving freshness problems more eagerly. However, care needs then to be taken to not remove freshness constraints from problems too early. For example, consider the following unification problem, which has no solution.

$$\{a \ \#?\ X, a \approx ?\ X\} \tag{14}$$

If one applies first (#?-suspension) followed by ($\approx$?-variable), then one gets a *wrong* result, namely $(\{a \ \#\ X\}, [X := a])$. The problem is that the substitution $[X := a]$ has not been properly propagated to the freshness constraint $a \ \#\ X$. If freshness problems are solved more eagerly, then proper propagation of substitutions into freshness constraints needs to be taken into account.

Remark 3.10 (Atoms are not variables). Nominal unification unifies variables, but it does not unify atoms. Indeed the operation of identifying two atoms by renaming one of them to be the other does not necessarily preserve the validity of the judgements in Figure 2. For example, $\emptyset \vdash a.b \approx c.b$ holds if $b \neq a, c$; but renaming b to be a in this judgement we get $\emptyset \vdash a.a \approx c.a$, which does not hold so long as $a \neq c$. Referring to Definition 2.3, you will see that we do allow variables ranging over sorts of atoms; and such variables can be unified like any other variables. However, if A is such a variable, then it cannot appear in abstraction position, i.e. as $A.t$. This is because we specifically restricted abstraction to range over atoms, rather than over arbitrary terms of atom sort. Such a restriction seems necessary to obtain single, most general, solutions to nominal unification problems. For without such a restriction, because of rule ($\approx$-abstraction-2) in Figure 2 we would also have to allow variables to appear on the left-hand side of freshness relations and in suspended permutations. So then we would get unification problems like $\{(A\,B){\cdot}C \approx ?\ C\}$, where A, B and C are variables of atom sort; this has two incomparable solutions, namely $(\emptyset, [A := B])$ and $(\{A \ \#\ C, B \ \#\ C\}, \varepsilon)$.

4 Related work

Higher-order pattern unification

Most previous work on unification for languages with binders is based on forms of higher-order unification, i.e. solving equations between λ-terms modulo $\alpha\beta\eta$-equivalence ($=_{\alpha\beta\eta}$) by capture-avoiding substitution of terms for function variables. Notable among that work is Miller's *higher-order pattern unification* used in his L_λ logic programming language [3]. This kind of unification retains the good properties of first-order unification: a linear-time decision procedure and existence of most general unifiers. This good behaviour of higher-order pattern unification is the result of equations being solved only modulo $=_{\alpha\beta_0\eta}$ (where β_0-equivalence is the restricted form of β-equivalence that identifies $(\lambda x.M)y$ and $M[y/x]$ with y being a variable) and of λ-terms being restricted such that function variables may only be applied to distinct bound variables. An empirical study by Michaylov and Pfenning [27] suggests that most unifications arising dynamically in higher-order logic programming satisfy Miller's restrictions, but that it rules out some useful programming idioms.

The main difference between higher-order pattern unification and nominal unification is that the former solves a set of equations by calculating a *capture-avoiding* substitution, while the latter calculates a *possibly-capturing* substitution *and* some freshness constraints. Moreover, unifiers in higher-order pattern unification solve equations with respect to $=_{\alpha\beta_0\eta}$; whereas in nominal unification, unifiers solve equations with respect to the equivalence $\approx$ defined in Figure 2, which agrees with α-equivalence on ground terms (see Proposition 2.16), but differs from it on open terms, since unlike α-equivalence, it is respected by possibly-capturing substitutions (see Lemma 2.14). For us, the main disadvantage of higher-order pattern unification is the one common to most approaches based on higher-order abstract syntax that was discussed in the Introduction: one cannot *directly* express the common idiom of possibly-capturing substitution of terms for metavariables. Instead one has to encode metavariables X as function variables applied to distinct lists of (bound) variables, $X\,x_1 \ldots x_n$, and use capture-avoiding substitution. At first sight, there seems to be a simple encoding for doing that. Consider for example the purely equational nominal unification problem

$$a.X \approx ?\; b.b \tag{15}$$

which is solved by $(\emptyset, [X := a])$. The literal encoding as the higher-order pattern unification problem $\lambda a.X =_{\alpha\beta_0\eta} ?\lambda b.b$ does not work of course, because there is no capture-avoiding substitution that solves this problem. However, X can be made dependent on a yielding the unification problem

$$\lambda a.(Xa) =_{\alpha\beta_0\eta} ?\; \lambda b.b \tag{16}$$

which is solved by the capture-avoiding substitution of $\lambda c.c$ for X. If one further applies to $\lambda c.c$ the atom a used by the encoding, then one can read back the original solution $[X := a]$ by applying some β-reductions. There are however several problems with this encoding. First, the encoding in general results in a quadratic blow-up in the size of terms. For example the nominal unification problem

$$a.b.\langle X, Y\rangle \approx ?\ a.b.\langle a, b\rangle \tag{17}$$

solved by the unifier $(\emptyset, [X := a, Y := b])$ needs to be encoded so that X and Y depend on both a and b. This gives the higher-order pattern problem

$$\lambda a.\lambda b.\langle X\,a\,b, Y\,a\,b\rangle =_{\alpha\beta_0\eta} ?\ \lambda a.\lambda b.\langle a, b\rangle\ . \tag{18}$$

In the general case, the encoding needs to make metavariables dependent on *all* atoms occurring in a nominal unification problem, regardless of whether they actually occur in an individual equational problem. For example, if X occurs elsewhere within the scope of abstractions of c, d, e and f, then X needs to be encoded as $(X\,a\,b\,c\,d\,e\,f)$ even though an individual equational problem might contain only a and b. Secondly, and more importantly, we cannot see how to encode our freshness constraints using this kind of higher-order patterns. (Note that in nominal unification, freshness constraints do not necessarily come from analysing abstractions, rather they can be chosen arbitrarily.)

A more promising target for a reduction of nominal unification to some form of higher-order pattern unification is $\lambda\sigma$, a λ-calculus with de-Bruijn indices and explicit substitutions. Dowek *et al* [28] present a version of higher-order pattern unification for $\lambda\sigma$ in which unification problems are solved, as in nominal unification, by textual replacements of terms for variables; however a "pre-cooking" operation ensures that the textual replacements can be faithfully related to capture-avoiding substitutions. It seems possible that the freshness (as well as the equational) problems of nominal unification can be encoded into higher-order pattern unification problems over $\lambda\sigma$, using a non-trivial translation involving the use of the shift operator and the introduction of fresh unification variables. The details of this encoding still remain to be investigated. Furthermore, it is not clear to us how to translate solutions obtained via the encoding back into solutions of the original nominal unification problem. But even if it turns out that it is possible to reduce nominal unification to the algorithm of Dowek *et al*, the calculations involved in translating our terms into $\lambda\sigma$ patterns and then using higher-order pattern unification seem far more intricate than our simple algorithm that solves nominal unification problems directly. The conclusion we draw is that an encoding of nominal unification problems into higher-order pattern unification problems (using de Bruijn indices and explicit substitutions) might be possible, but such an encoding is no substitute in practice for having the simple, direct algorithm we presented here.

Hamana's β_0-unification of λ-terms with "holes"

Hamana [5,29] manages to add possibly-capturing substitution to a language like Miller's L_λ. This is achieved by adding syntax for explicit renaming operations and by recording implicit dependencies of variables upon bindable names in a typing context. The mathematical foundation for Hamana's system is the model of binding syntax of Fiore *et al* [24]. The mathematical foundation for our work appeared at the same time (see [10]) and is in a sense complementary. For in Hamana's system the typing context restricts which terms may be substituted for a variable by giving a finite set of names that *must contain* the free names of such a term; whereas we give a finite set of names which the term's free variables *must avoid*. Since α-conversion is phrased in terms of avoidance, i.e. freshness of names, our approach seems more natural if one wants to compute α-equivalences concretely. On top of that, our use of name permutations, rather than arbitrary renaming functions, leads to technical simplifications. In any case, the bottom line is that Hamana's system seems more complicated than the one presented here and does not possess most general unifiers.

Qu-Prolog

The work [8,9] on unification in Qu-Prolog is most closely related to that reported here. Qu-Prolog is a mature logic programming language addressing many problems we set out in the Introduction. To begin with, Qu-Prolog's unification algorithm unifies terms modulo α-equivalence and may produce solutions that, as in nominal unification, depend on freshness constraints (in Qu-Prolog such constraints are represented by a predicate called `not_free_in`). Furthermore, metavariables are substituted in a possibly-capturing manner. However, there are also a number of differences between nominal unification and unification in Qu-Prolog. The most obvious difference is that the term language in Qu-Prolog is richer than our term language over nominal signatures; for example Qu-Prolog allows variables in binding position and permits explicit substitutions of terms for variables. This richness of the term language leads to a number of difficulties. First, the unification problems in Qu-Prolog are only semi-decidable (whereas the nominal unification problems are decidable) and as a result the algorithm employed in Qu-Prolog can leave as unsolved some unification problems that are "too difficult". This means the unification transformations in Qu-Prolog, while shown not to delete any solutions nor to introduce any new ones, do not always lead to problems from which an explicit solution can be obtained. Secondly, as we illustrated in Remark 3.10, the possibility of forming terms with unification variables in binding position means that most general solutions may not exist.

Another difference arises from the fact that in Qu-Prolog binders are renamed via capture-avoiding substitutions. This means that fresh names need to be introduced during unification in order to respect α-equivalence. This is not necessary in nominal unification, because the permutation operation already respects α-equivalence. In fact the introduction of fresh atoms during unification leads to a more complicated notion of most general solution. Consider the following variant of the ($\approx$?-abstraction-2) transformation:

$$(\approx?\text{-abstraction-2}')\ \{a.t \approx?\ a'.t'\} \uplus P \overset{\varepsilon}{\Longrightarrow} \{(a\,b)\cdot t \approx?\ (a'\,b)\cdot t', b \ \#?\ t, b \ \#?\ t'\} \cup P$$

which is applicable provided $a \neq a'$ and b is a fresh atom, not occurring elsewhere in the problem. This rule is essentially the refinement step that unifies two abstracted terms in Qu-Prolog (see [8, Page 105]). If we were to use ($\approx$?-abstraction-2$'$) instead of ($\approx$?-abstraction-2) in our nominal unification algorithm, then when applied to the problem

$$\{\ a.X \approx?\ b.Y\ \} \tag{19}$$

it would produce the solution $(\{a \ \#\ Y, c \ \#\ Y\}, [X := (a\,c)(b\,c)\cdot Y])$. While this solution solves the problem, it is not a most general solution according to Definition 3.1—we lost the information that c is a completely fresh atom. On the other hand, applying transformation ($\approx$?-abstraction-2) to (19) leads to $(\{a \ \#\ Y\}, [X := (a\,b)\cdot Y])$—a most general solution.

Overall, the theory of Qu-Prolog's unification is more complex than that of nominal unification: in nominal unification we do not need to resort to a semantic notion of α-equivalence in order to show the correctness of the nominal unification algorithm; and the use of permutations makes our $\approx$-relation much simpler compared with Qu-Prolog's use of the traditional notion of α-equivalence extended to terms with metavariables.

5 Conclusion and Future Work

In this paper we have proposed a new solution to the problem of computing possibly-capturing substitutions that unify terms involving binders up to α-conversion. To do so we considered a many-sorted first-order term language with distinguished collections of constants called *atoms* and with *atom-abstraction* operations for binding atoms in terms. This provides a simple, but flexible, framework for specifying binding operations and their scopes, in which the bound entities are explicitly named. By using variables prefixed with suspended permutations, one can have substitution of terms for variables both allow capture of atoms by binders and respect α-equivalence (renaming of bound atoms). The definition of α-equivalence for the term language makes use of an auxiliary *freshness* relation between atoms and terms which generalises the "not a free atom of" relation from ground terms to terms with variables; furthermore, because variables stand for unknown terms,

```
type Gamma (var X) A :- mem (pair X A) Gamma.
type Gamma (app M N) B :- type Gamma M (arrow A B),
                          type Gamma N A.
type Gamma (lam x.M) (arrow A B) / x#Gamma :-
                          type (pair x A)::Gamma M B.

mem A A::Tail.
mem A B::Tail :- mem A Tail.
```

Fig. 5. An example αProlog program

hence with unknown free atoms, it is necessary to make hypotheses about the freshness of atoms for variables in judgements about term equivalence and freshness. This reliance on "freshness", coupled with name-swapping rather than renaming, lead to a new notion of unification problem in which instances of both equivalence and freshness have to be solved by giving term-substitutions and (possibly) freshness conditions on variables in the solution. We showed that this unification problem is decidable and unitary.

Cheney, Gabbay and Urban [30,31] are investigating the extent to which nominal unification can be used in resolution-based proof search for a form of first-order logic programming for languages with binders (with a view to providing better machine-assistance for structural operational semantics). Such a logic programming language should permit a concrete, "nominal" approach to bound entities in programs while ensuring that computation (which in this case is the computation of answers to queries) respects α-equivalence between terms. This is illustrated with the Prolog-like program in Figure 5, which implements a simple typing algorithm for λ-terms. The third clause is the interesting one. First, note the term `(lam x.M)`, which unifies with any λ-abstraction. The binder `x`, roughly speaking, has in the "nominal" approach a value which can be used in the body of the clause, for example for adding `(pair x A)` to the context `Gamma`. Secondly, the freshness constraint `x # Gamma` ensures that `Gamma` cannot be replaced by a term that contains `x` freely. Since this clause is intended to implement the usual rule for typing λ-abstractions

$$\frac{\{x : A\} \cup \Gamma \ \triangleright\ M : B}{\Gamma \ \triangleright\ \lambda x.M : A \supset B}$$

its operational behaviour is given by: choose fresh names for `Gamma`, `x`, `M`, `A` and `B` (this is standard in Prolog-like languages), unify the head of the clause with the goal formula, apply the resulting unifier to the body of the clause and make sure that `Gamma` is not replaced by a term that contains freely the fresh name we have chosen for `x`. Similar facilities for *functional programming* already exist in the FreshML language, built upon the same foundations: see [13] and www.freshml.org. We are also interested in the special case of "nominal matching" and its application to term-rewriting modulo α-equivalence.

A note on complexity

If these applications show that nominal unification is practically useful, then it becomes important to study its complexity. The presentations of the term language in Section 2 and of the algorithm in Section 3 were chosen for clarity and to make the proof of correctness [3] easier, rather than for efficiency. One source of increased efficiency is to delay the application of permutations: instead of pushing permutation inside terms until they reach suspension as we do here, one should just push them under the first constructor (pairing, function symbol application, or atom-abstraction) in order to proceed with the next step of decomposition. However, the main inefficiency of the algorithm presented in Section 3 comes from the lack of sharing in terms and substitutions. Thus the unification problem taken from [32]

$$\{f(X_1, X_1) \approx ?\ X_2, f(X_2, X_2) \approx ?\ X_3, \ldots, f(X_{n-1}, X_{n-1}) \approx ?\ X_n\}$$

which illustrates that the naïve algorithm for classical first-order unification has exponential time complexity, also applies to the algorithm for nominal unification given here. If one adapts a representation for terms using techniques developed in [32] or [33], which are based on directed acyclic graphs, then one easily arrives at an algorithm with quadratic time complexity. The reason for the quadratic, rather than linear, time-complexity is that permutations need to be applied to some atoms when deciding whether the rules ($\approx$?-abstraction-1) or ($\approx$?-abstraction-2) are applicable, and these permutations (represented as lists of swappings) might grow linearly with the number of nodes. Using a representation of permutations that allows for a more efficient calculation of their action on atoms does not improve the quadratic time complexity, because it makes the operation of composing two permutations become linear, while this can be done in constant time when using the list-of-swappings representation. For higher-order patterns, Qian managed to developed a unification algorithm with linear time-complexity [34]. It seems that adapting Qian's algorithm to nominal unification via an encoding of nominal terms into higher-order patterns as discussed in Section 4 will not solve this problem. For the encoding makes the resulting higher-order patterns quadratically longer than the original nominal terms, so this method would only provide another algorithm with quadratic time complexity.

To sum up, there is a version of nominal unification with quadratic time complexity, but is it is still an open question whether a version can be developed with *linear* time complexity.

[3] See `http://www.cl.cam.ac.uk/users/cu200/Unification` for the Isabelle proof scripts.

Acknowledgements

A preliminary version of this paper appeared as [35]. We thank James Cheney, Gilles Dowek, Roy Dyckhoff, Dale Miller, Frank Pfenning, Francois Pottier and Helmut Schwichtenberg for comments on this work. This research was supported by UK EPSRC grants GR/R29697 (Urban) and GR/R07615 (Pitts and Gabbay).

References

[1] G. Dowek, T. Hardin, C. Kirchner, Higher-order unification via explicit substitutions, in: 10th Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, Washington, 1995, pp. 366–374.

[2] C. A. Gunter, Semantics of Programming Languages: Structures and Techniques, Foundations of Computing, MIT Press, 1992.

[3] D. Miller, A logic programming language with lambda-abstraction, function variables, and simple unification, Journal of Logic and Computation 1 (1991) 497–536.

[4] G. Dowek, Higher-order unification and matching, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Elsevier, 2001, Ch. 16, pp. 1009–1062.

[5] M. Hamana, A logic programming language based on binding algebras, in: N. Kobayashi, B. C. Pierce (Eds.), Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings, Vol. 2215 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2001, pp. 243–262.

[6] M. Hashimoto, A. Ohori, A typed context calculus, Theoretical Computer Science 266 (2001) 249–271.

[7] M. Sato, T. Sakurai, Y. Kameyama, A simply typed context calculus with first-class environments, Journal of Functional and Logic Programming 2002 (4).

[8] P. Nickolas, P. J. Robinson, The Qu-Prolog unification algorithm: Formalisation and correctness, Theoretical computer Science 169 (1996) 81–112.

[9] R. Paterson, Unification of schemes of quantified terms, in: Proc. of UNIF 1990, 1990, unpublished proceedings.

[10] M. J. Gabbay, A. M. Pitts, A new approach to abstract syntax with variable binding, Formal Aspects of Computing 13 (2002) 341–363.

[11] L. Caires, L. Cardelli, A spatial logic for concurrency (part II), in: L. Brim, P. Jančar, M. Křetínský, A. Kučera (Eds.), CONCUR 2002 – Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002. Proceedings, Vol. 2421 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2002, pp. 209–225.

[12] L. Cardelli, P. Gardner, G. Ghelli, Manipulating trees with hidden labels, in: A. D. Gordon (Ed.), Foundations of Software Science and Computation Structures, 6th International Conference, FOSSACS 2003, Warsaw, Poland. Proceedings, Vol. 2620 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 216–232.

[13] M. R. Shinwell, A. M. Pitts, M. J. Gabbay, FreshML: Programming with binders made simple, in: Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden, ACM Press, 2003, pp. 263–274.

[14] A. Salibra, On the algebraic models of lambda calculus, Theoretical Computer Science 249 (2000) 197–240.

[15] F. Honsell, M. Miculan, I. Scagnetto, An axiomatic approach to metareasoning on nominal algebras in HOAS, in: F. Orejas, P. G. Spirakis, J. Leeuwen (Eds.), 28th International Colloquium on Automata, Languages and Programming, ICALP 2001, Crete, Greece, July 2001. Proceedings, Vol. 2076 of Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 2001, pp. 963–978.

[16] A. M. Pitts, Nominal logic, a first order theory of names and binding, Information and Computation 186 (2003) 165–193.

[17] M. Sato, T. Sakurai, Y. Kameyama, A. Igarashi, Calculi of meta-variables, in: M. Baaz (Ed.), Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria. Proccedings, Vol. 2803 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 484–497.

[18] A. Martelli, U. Montanari, An efficient unification algorithm, ACM Trans. Programming Languages and Systems 4 (2) (1982) 258–282.

[19] J. W. Klop, Term rewriting systems, in: S. Abramsky, D. M. Gabbay, T. S. E. Maibaum (Eds.), Handbook of Logic in Computer Science, Volume 2, Oxford Univerity Press, 1992, pp. 1–116.

[20] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[21] H. P. Barendregt, The Lambda Calculus: its Syntax and Semantics, North-Holland, 1984.

[22] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes (parts I and II), Information and Computation 100 (1992) 1–77.

[23] G. D. Plotkin, An illative theory of relations, in: R. Cooper, Mukai, J. Perry (Eds.), Situation Theory and its Applications, Vol. 22 of CSLI Lecture Notes, Stanford University, 1990, pp. 133–146.

[24] M. P. Fiore, G. D. Plotkin, D. Turi, Abstract syntax and variable binding, in: 14th Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, Washington, 1999, pp. 193–202.

[25] R. Milner, M. Tofte, R. Harper, D. MacQueen, The Definition of Standard ML (Revised), MIT Press, 1997.

[26] F. Pfenning, C. Elliott, Higher-order abstract syntax, in: Proc. ACM-SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 1988, pp. 199–208.

[27] S. Michaylov, F. Pfenning, An empirical study of the runtime behaviour of higher-order logic programs, in: D. Miller (Ed.), Proc. Workshop on the λProlog Programming Language, University of Pennsylvania, 1992, pp. 257–271, CIS Technical Report MS-CIS-92-86.

[28] G. Dowek, T. Hardin, C. Kirchner, F. Pfenning, Higher-order unification via explicit substitutions: the case of higher-order patterns, in: Proc. of JICSLP, 1996, pp. 259–273.

[29] M. Hamana, Simple β_0-unification for terms with context holes, in: C. Ringeissen, C. Tinelli, R. Treinen, R. M. Verma (Eds.), Proc. of UNIF 2002, 2002, unpublished proceedings.

[30] J. Cheney, C. Urban, αProlog, a fresh approach to logic programming modulo α-equivalence, in: J. Levy, M. Kohlhase, J. Niehren, M. Villaret (Eds.), Proc. of UNIF 2003, no. DSIC-II/12/03 in Departamento de Sistemas Informáticos y Computación Technical Report Series, Universidad Politécnica de Valencia, 2003, pp. 15–19.

[31] M. Gabbay, J. Cheney, A proof theory for nominal logic, in: Nineteenth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, Washington, 2004.

[32] M. S. Paterson, M. N. Wegman, Linear unification, Journal of Computer System Sciences 16 (2) (1978) 158–167.

[33] A. Martelli, U. Montanari, An efficient unification algorithm, ACM Transactions on Programming Languages and Systems 4 (2).

[34] Z. Qian, Unification of higher-order patterns in linear time and space, Journal of Logic and Computation 6 (3) (1996) 315–341.

[35] C. Urban, A. M. Pitts, M. J. Gabbay, Nominal unification, in: M. Baaz (Ed.), Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria. Proceedings, Vol. 2803 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 513–527.